Theoretical Computer Science 411 (2010) 986–994

Contents lists available at ScienceDirect

Theoretical Computer Science

journal homepage: www.elsevier.com/locate/tcs

Efficient algorithms for the sum selection problem and k maximum sums problem☆

Tien-Ching Lin [a,*], D.T. Lee [a,b]

[a] Institute of Information Science, Academia Sinica, Nankang, Taipei 115, Taiwan

[b] Department of Computer Science and Information Engineering, National Taiwan University, Taipei, Taiwan

ARTICLE INFO

Article history:
Received 8 June 2009
Accepted 7 November 2009
Communicated by D.-Z. Du

Keywords:
k maximum sums problem
Sum selection problem
Maximum sum problem
Maximum sum subarray problem

ABSTRACT

Given a sequence of n real numbers $A = a_1, a_2, \ldots, a_n$ and a positive integer k, the Sum Selection Problem is to find the segment $A(i^*, j^*) = a_{i^*}, a_{i^*+1}, \ldots, a_{j^*}$ such that the rank of the sum $s(i^*, j^*) = \sum_{t=i^*}^{j^*} a_t$ is k over all $\frac{n(n-1)}{2}$ segments. We present a deterministic algorithm for this problem that runs in $O(n \log n)$ time. The previously best known result for this problem is a randomized algorithm that runs in expected $O(n \log n)$ time. Applying this algorithm we can obtain a deterministic algorithm for the k Maximum Sums Problem, i.e., the problem of enumerating the k largest sum segments, that runs in $O(n \log n + k)$ time. The previously best known randomized and deterministic algorithms for the k Maximum Sums Problem run respectively in expected $O(n \log n + k)$ time and in worst case $O(n \log^2 n + k)$ time.

© 2009 Elsevier B.V. All rights reserved.

1. Introduction

Consider a sequence of n real numbers $A = a_1, a_2, \ldots, a_n$. A segment $A(i, j) = a_i, a_{i+1}, \ldots, a_j$ is a consecutive subsequence of A. Given a sequence of n real numbers A, the Maximum Sum Problem is to find the segment $A(i^*, j^*) = a_{i^*}, a_{i^*+1}, \ldots, a_{j^*}$ whose sum $s(i^*, j^*) = \sum_{t=i^*}^{j^*} a_t$ is the maximum among all possible segments. This problem, also called Maximum Sum Subsequence Problem, was first introduced by Bentley [7,8] and can be easily solved in $O(n)$ time [8,14].

The two-dimensional counterpart is the Maximum Sum Subarray Problem, which is to find the submatrix of a given $m \times n$, $m \leq n$, matrix of real numbers, the sum of whose entries is the maximum among all $O(m^2n^2)$ submatrices. The problem was solved in $O(m^2n)$ time [8,14,19]. Tamaki and Tokuyama [21] gave the first sub-cubic time algorithm for this problem and Takaoka [20] later gave a simplified algorithm achieving sub-cubic time as well. Many parallel algorithms under different parallel models of computation were also obtained [3,17–19]. The Maximum Sum Problem has many applications in pattern recognition, image processing and data mining [1,13].

A natural generalization of the above Maximum Sum Problem is the k Maximum Sums Problem which is to find the k segments such that their sums are the k largest over all $\frac{n(n-1)}{2}$ segments. Bae and Takaoka [4] presented an $O(kn)$ time algorithm for this problem. Bengtsson and Chen [6] gave an $O(\min\{k + n \log^2 n, n\sqrt{k}\})$ time algorithm, or $O(n \log^2 n + k)$ time in the worst case. Cheng et al. [10] and Bae and Takaoka [5] recently gave an $O(n + k \log(min\{n, k\}))$ time algorithm and an $O((n + k) \log k)$ time algorithm respectively, which is superior to Bengtsson and Chen's when k is $o(n \log n)$, but both run

☆ Research supported in part by the National Science Council under the Grant Nos. NSC-94-2213-E-001-004, NSC-95-2221-E-001-016-MY3, and NSC 94-2752-E-002-005-PAE, and by the Taiwan Information Security Center (TWISC), National Science Council under the Grant No. NSC94-3114-P-001-001-Y.

* Corresponding address: National Taiwan University, Department of Computer Science and Information Engineering, Taipei, Taiwan. Tel.: +886 2 2788 3799x1602; fax: +886 2 2782 4814.

E-mail addresses: kero@iis.sinica.edu.tw (T.-C. Lin), dtlee@iis.sinica.edu.tw (D.T. Lee).

0304-3975/$ – see front matter © 2009 Elsevier B.V. All rights reserved.
doi:10.1016/j.tcs.2009.11.005

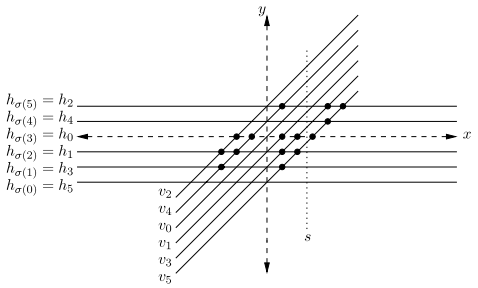

Fig. 1. Given $A = a_1, a_2, a_3, a_4, a_5 = 1, -3, 4, -3, 4$, we have $S = \{s_0, s_1, s_2, s_3, s_4, s_5, s_6\} = \{0, 1, -2, 2, -1, 3\}$, $H = \{h_i | h_i : y = -s_i, i = 0, 1, \ldots, 6\}$ and $V = \{v_i | v_i : y = x - s_i, i = 0, 1, \ldots, 6\}$ respectively. The intersection points shown in dark solid dots are feasible and others are infeasible.

in $O(n^2 \log n)$ time in the worst case. Lin and Lee [15] recently gave an expected $O(n \log n + k)$ time randomized algorithm based on a randomized algorithm which finds in expected $O(n \log n)$ time the segment whose sum is the kth smallest by using a random sampling technique, for any given positive integer $1 \leq k \leq \frac{n(n-1)}{2}$. The latter problem is referred to as the Sum Selection Problem. In this paper we will give a deterministic $O(n \log n + k)$ time algorithm for the k Maximum Sums Problem based on a deterministic $O(n \log n)$ time algorithm for the Sum Selection Problem.

The rest of this paper is organized as follows. Section 2 gives a deterministic algorithm for the Sum Selection Problem. Section 3 gives a deterministic algorithm for the k Maximum Sums Problem. Section 4 gives some conclusion.

2. Algorithm for Sum Selection Problem

We define the *rank* $r(x, P)$ of an element x in a set $P \subseteq \mathbf{R}$ of real numbers to be the number of elements in P no greater than x, i.e. $r(x, P) = |\{y | y \in P, y \leq x\}|$. Given a sequence A of real numbers $a_1, a_2, \ldots, a_n$, and a positive integer $1 \leq k \leq \frac{n(n-1)}{2}$, the Sum Selection Problem is to find the segment $A(i^*, j^*)$ over all $\frac{n(n-1)}{2}$ segments such that the rank of the sum $s(i^*, j^*) = \sum_{t=i^*}^{j^*} a_t$ in the set of possible subsequence sums is k. That is, we would like to find $s^* = s(i^*, j^*)$ for some $i^* < j^*$ such that $r(s^*, P) = k$ where $P = \{s(i, j) | s(i, j) = \sum_{t=i}^{j} a_t, 1 \leq i \leq j \leq n\}$.

We will transform the Sum Selection Problem into an intersection selection problem in a line arrangement in $O(n)$ time as follows. We first define the set $S = \{s_0, s_1, \ldots, s_n\}$, where each $s_i = \sum_{t=1}^{i} a_t$, $i = 1, 2, \ldots, n$, is the prefix sum of sequence A, and $s_0 = 0$. We then define a line arrangement $\mathcal{A}(H \cup V)$ in the plane, consisting of two sets of lines $H = \{h_i | h_i : y = -s_i, i = 0, 1, \ldots, n\}$ and $V = \{v_i | v_i : y = x - s_i, i = 0, 1, \ldots, n\}$. For any two lines $h_i \in H$ and $v_j \in V$, they intersect at the point $p_{ij} = (x_{ij}, y_{ij})$ with abscissa $x_{ij} = s_j - s_i$. Therefore, the abscissa of the intersection point of any two lines $h_i \in H$ and $v_j \in V$ for $i < j$ is equal to the sum $s(i + 1, j)$ of the segment $A(i + 1, j)$. We say that an intersection point of two lines $h_i \in H$ and $v_j \in V$ is *feasible* if $i < j$, and *infeasible*, otherwise. Note that there are totally n^2 intersection points in the line arrangement $\mathcal{A}(H \cup V)$, in which $\frac{n(n-1)}{2}$ of them are feasible, and $\frac{n(n+1)}{2}$ are infeasible. An example of the line arrangement $\mathcal{A}(H \cup V)$ is shown in Fig. 1. Let X_f denote the set of abscissae of feasible intersection points, i.e., $X_f = \{x_{ij} | p_{ij} = (x_{ij}, y_{ij})$ is a feasible intersection point of $\mathcal{A}(H \cup V)\}$. The Sum Selection Problem is equivalent to the following intersection selection problem.

> Given a line arrangement $\mathcal{A}(H \cup V)$ in $\mathbf{R}^2$, $H = \{h_0, h_1, \ldots, h_n\}$ and $V = \{v_0, v_1, \ldots, v_n\}$, where $h_i : y = -s_i$ and $v_j : y = x - s_j$, and an integer $k > 0$, find the feasible intersection point $p_{i^*j^*} = (x_{i^*j^*}, y_{i^*j^*})$ such that $r(x_{i^*j^*}, X_f) = k$.

Given a set of n lines in the plane and an integer k, $1 \leq k \leq \frac{n(n-1)}{2}$, the well-known dual problem of the Slope Selection Problem[1] in computational geometry is to find the intersection point whose x-coordinate is the kth smallest among all intersection points of these n lines. Cole et al. [11] developed an approximate counting scheme combining the AKS sorting network and parametric search to obtain an optimal $O(n \log n)$ algorithm for this problem. Brönnimann and Chazelle [9] modified their approximate counting scheme combining ε-net to obtain yet another optimal algorithm for this problem. The Sum Selection Problem can be viewed as a variant of the Slope Selection Problem. Since we do not know how many infeasible intersection points of $\mathcal{A}(H \cup V)$ are to the left of the kth feasible intersection point, and thus we do not know the actual rank of the kth feasible intersection point in the set of all intersection points of $\mathcal{A}(H \cup V)$. The actual rank of the kth feasible intersection point may lie between k and $k + \frac{n(n+1)}{2}$ in the set of all intersection points of $\mathcal{A}(H \cup V)$. Therefore, we cannot solve the Sum Selection Problem by fixing some specific rank and directly applying the slope selection algorithms [9,11]. We will give a deterministic algorithm for this problem that runs in $O(n \log n)$ time combining the parametric search technique of Megiddo [16], AKS sorting network [2] and a *new* approximate counting scheme. This new approximate counting scheme can be thought of as a generalization of the approximate counting schemes developed

[1] Given a set of n points in the plane and an integer k, $1 \leq k \leq \frac{n(n-1)}{2}$, the slope selection problem is to select the pair of points determining the line with the kth smallest slope.

by Cole et al. [11] and Brönnimann and Chazelle [9]. In the following, we will omit $H \cup V$ in the notation $\mathcal{A}(H \cup V)$ and simply use $\mathcal{A}$ to refer to the line arrangement.

Given a vertical line $v : x = s$, the number of intersection points of $\mathcal{A}$ on the line v at $x = s$ or to its left is denoted $\lambda(s)$ and the number of feasible intersection points is denoted $\lambda_f(s)$. The vertical order of the intersections of the line v at $x = s$ and the line arrangement $\mathcal{A}$ defines a *permutation* $\pi(s)$ of $H \cup V$ at s with $\pi(-\infty)$ being the identity permutation. An example of $\pi(s) = (h_5, h_3, h_1, v_5, h_0, v_3, h_4, v_1, h_2, v_0, v_4, v_2)$ and $\pi(-\infty) = (v_5, v_3, v_1, v_0, v_4, v_2, h_5, h_3, h_1, h_0, h_4, h_2)$ is shown in Fig. 1. An *inversion* of a permutation $(p_1, p_2, \ldots, p_n)$ of the identity permutation $(1, 2, \ldots, n)$ is a pair of indices $i < j$ with $p_i > p_j$. It is easy to see that the number of inversions, denoted by $I(\pi(s))$, of a permutation $\pi(s)$ is exactly $\lambda(s)$. Similarly, the number of feasible inversions, denoted by $I_f(\pi(s))$, of $\pi(s)$ is $\lambda_f(s)$. Therefore, the Sum Selection Problem is also equivalent to finding some s^* such that $I_f(\pi(s^*)) = k$.

The problem of finding s^* can be viewed as an unusual sorting problem attempting to sort the set of lines $H \cup V$ at $x = s^*$ without knowing the value of s^*, i.e. to sort $h_0(s^*), v_0(s^*), h_1(s^*), v_1(s^*), \ldots, h_n(s^*), v_n(s^*)$ in vertical order without knowing the value of s^*. We know that this sorting problem may be achieved in $O(n \log n)$ comparisons. In particular, the comparisons of the forms "$h_i(s^*) \leq h_j(s^*)$" and "$v_i(s^*) \leq v_j(s^*)$" can be solved in $O(n \log n)$ time by any usual optimal sorting algorithm, since the ordering of h_i's, which is identical to that of v_j's, is independent of s^*. However, the comparison of the form "$h_i(s^*) \leq v_j(s^*)$", can be answered by a counting subroutine (Lemma 1) which, given any vertical line $x = s$, can quickly compute $\lambda_f(s)$, the number of feasible intersection points of $\mathcal{A}$ that lie on it or to its left in $O(n \log n)$ time. That is, we first find x_{ij}, the x-coordinate of intersection point of h_i and v_j in constant time and call the counting subroutine with $s_\ell = -\infty$ and $s_r = x_{ij}$. If the return value of the subroutine is less than or equal to k, we get $s^* \geq x_{ij}$ and $h_i(s^*) \leq v_j(s^*)$. Otherwise we get $s^* < x_{ij}$ and $h_i(s^*) > v_j(s^*)$. After solving the unusual sorting problem we can obtain the permutation $\pi(s^*)$ without knowing the value of s^*, and obtain $s^* = \max\{x_{\pi(s^*)[i]\pi(s^*)[i+1]}\}$.

Lemma 1 (*[15], Lemma 2*). *Given a sequence A of n real numbers $a_1, a_2, \ldots, a_n$ and two real numbers s_ℓ, s_r with $s_\ell \leq s_r$, it takes $O(n)$ space and $O(n \log n)$ time to count the total number of segments $A(i, j)$, $1 \leq i \leq j \leq n$, among all $\frac{n(n-1)}{2}$ segments such that their sums $s(i, j)$ satisfy $s_\ell \leq s(i, j) \leq s_r$.*

We now describe how to solve the unusual sorting problem. We will use the parametric search approach running a sequential simulation of a generic parallel sorting algorithm, which attempts to sort the arrangement of lines at $x = s^*$, where s^* is the x-coordinate of the desired kth leftmost feasible intersection point, without knowing the value of s^*. A naive algorithm is to use a parallel sorting algorithm of depth $O(\log n)$ and $O(n)$ processors developed by Ajtai, Komlós, and Szemerédi [2], and at each parallel step we may perform $\frac{n}{2}$ comparisons between pairs of lines. Since each comparison can be solved in $O(n \log n)$ time and $O(n)$ space following Lemma 1, it takes $O(n^2 \log n)$ time at each parallel step, and $O(n^2 \log^2 n)$ time overall.

However, we can improve it by the following slightly complicated algorithm. That is, we compute the median x_m of the x-coordinates of all the intersection points of these $\frac{n}{2}$ pairs of lines in each parallel step, and call the counting subroutine with $s_\ell = -\infty$ and $s_r = x_m$, which can answer half of the comparisons in $O(n \log n)$ time. For the $\frac{n}{4}$ unresolved comparisons at the same step, we again find the median x'_m among these $\frac{n}{4}$ x-coordinates and call the counting subroutine with $s_\ell = -\infty$ and $s_r = x'_m$, which can answer half of these $\frac{n}{4}$ unresolved comparisons in $O(n \log n)$ time. Repeating the above *binary search* process $O(\log n)$ times we can answer all $\frac{n}{2}$ comparisons in $O(n \log^2 n)$ time in each parallel step. We thus obtain an algorithm that runs in $O(n \log^3 n)$ time.

We can further improve $O(n \log^3 n)$ to $O(n \log^2 n)$ by using a well-known technique due to Cole [12] as follows. Instead of invoking $O(\log n)$ counting subroutine calls at each parallel step to resolve all comparisons at this step, we call the counting subroutine only a constant number of times. Although this does not resolve all comparisons of this parallel step, but it does resolve a large fraction of them. All the unresolved comparisons at this step will be deferred to the next parallel step. Suppose that each of the unresolved comparisons can affect only a constant number of comparisons executed at the next parallel step. Each parallel step is now a mixture of many parallel steps. Cole shows that if it is implemented carefully by assigning an appropriate time-dependent weight to each unresolved comparison and choosing the weighted median at each step of the binary search, the number of the parallel steps of the algorithm increases only by an additive $O(\log n)$ steps. Since each of these steps uses only a constant number of counting subroutine calls, the whole running time improves to $O(n \log^2 n)$.

The final step to improve the sum selection algorithm from $O(n \log^2 n)$ to $O(n \log n)$ is to develop an approximate counting scheme. Note that the expensive counting subroutine, Lemma 1, can be used not only to find $\lambda_f(s)$ for each s given by the sorting network but also to determine the relative ordering of s and s^* in $O(n \log n)$ time.

Instead of invoking the expensive counting subroutines $O(\log n)$ times, we shall develop an approximate counting scheme, that counts the number of feasible inversions of desired permutations only approximately, with an error that gets smaller and smaller as we get closer to the desired s^*. The idea of the approximate counting scheme is to use an approximation algorithm to approximate $I_f(\pi(s))$ in $O(n)$ time for each s chosen by the sorting network. If the error of $I_f(\pi(s))$ for the approximation algorithm is small enough, then we can decide the relative ordering of s and s^* directly. Otherwise, we will refine the approximation until we can decide the relative ordering of s and s^*. It turns out that an amortized $O(n \log n)$ extra time is sufficient to refine approximations throughout the entire course of the algorithm.

We will modify the approximate counting scheme presented in [9,11] for solving the slope selection problem. We define an m-block *left-compatible* permutation $\pi_l(s)$ of $\pi(s)$ and an m-block *right-compatible* permutation $\pi_r(s)$ of $\pi(s)$ as follows:

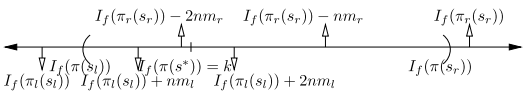

Fig. 2. The sum selection algorithm maintains an interval (s_l, s_r) containing s^* satisfying invariant conditions $(I1)$ and $(I2)$.

The permutation $\pi(s)$ will be partitioned into a collection of at most $2n/m$ blocks, each containing at most m lines consecutive in $\pi(s)$. For any two lines h_i, h_j in H, we say that $h_i < h_j$ if $-s_i < -s_j$. Let $(\sigma(0), \sigma(1), \ldots, \sigma(n))$ denote the permutation of $\{0, 1, \ldots, n\}$ such that $h_{\sigma(0)}, h_{\sigma(1)}, \ldots, h_{\sigma(n)}$ are in ascending vertical order, i.e. $h_{\sigma(0)} < h_{\sigma(1)} < \cdots < h_{\sigma(n)}$. We define $S = \{S_0, S_1, \ldots, S_q\}$, $\frac{n}{m} \leq q \leq \frac{2n}{m}$, to be an *m-block* of H at s, each of S_t containing between $m/2$ and m elements of H, such that it has the property that if $h_a \in S_i$, $h_b \in S_j$ and $i < j$, then $h_a < h_b$. Let $l(S_t)$ be the smallest element in S_t for each t, and $l(S_{q+1}) = \infty$. We also define $T = \{T_0, T_1, \ldots, T_q\}$ to be an *m-block* of V at s, where $T_t = \{v_j \in V | l(S_t) \leq v_j(s) < l(S_{t+1})\}$. We then define $\pi_l(s) = (T'_0, S'_0, T'_1, S'_1, \ldots, T'_q, S'_q)$ is an m-block *left-compatible* permutation $\pi(s)$ and $\pi_r(s) = (S'_0, T'_0, S'_1, T'_1, \ldots, S'_q, T'_q)$ is an m-block *right-compatible* permutation of $\pi(s)$, where S'_i and T'_i are each an ordered set of S_i and T_i respectively. It is not difficult to see that the following inequalities hold for both $\pi_l(s)$ and $\pi_r(s)$.

$$I_f(\pi_l(s)) \leq I_f(\pi(s)) \leq I_f(\pi_l(s)) + mn.$$

$$I_f(\pi_r(s)) - mn \leq I_f(\pi(s)) \leq I_f(\pi_r(s)).$$

We shall maintain left-compatible and right-compatible permutations of $\pi(s)$ to give a good approximation on the number of feasible inversions of the permutation with the property that the smaller the block size m, the finer the approximation.

We now give an $O(n \log n)$ algorithm for the Sum Selection Problem as follows. We assume, for simplicity, that $n = 2^r$ for some integer r and the all the fractions in this algorithm are taken by floor or ceiling functions, as appropriate. We define $sign(s)$ to be 1 if s is a positive and -1 if s is negative. The algorithm maintains an interval (s_l, s_r) containing s^*, an m_l-block left-compatible permutation $\pi_l(s_l)$ at s_l and an m_r-block right-compatible permutation $\pi_r(s_r)$ at s_r such that they satisfy invariant conditions $(I1)$ and $(I2)$ below. An example of an interval (s_l, s_r) containing s^* is shown in Fig. 2.

$$(I1)\ I_f(\pi_l(s_l)) + m_l n \leq I_f(\pi(s^*)) \leq I_f(\pi_r(s_r)) - m_r n.$$

$$(I2)\ I_f(\pi_r(s_r)) - 2m_r n \leq I_f(\pi(s^*)) \leq I_f(\pi_l(s_l)) + 2m_l n.$$

$(I1)$ ensures that s^* lies within the interval (s_l, s_r). This follows from the fact that $I_f(\pi(s_l)) \leq I_f(\pi_l(s_l)) + m_l n \leq I_f(\pi(s^*)) \leq I_f(\pi_r(s_r)) - m_r n \leq I_f(\pi(s_r))$. $(I2)$ is to ensure that the left-compatible and right-compatible permutations are no finer than needed.

If $k < n$, then we can solve the sum selection problem by using the algorithm due to Cheng et al. [10]. Let us assume $k \geq n$ in the following. To initialize the algorithm, we set $m_l = \frac{k}{n}$, $s_l = -\infty$, $\pi_l(s_l) = (v_{\sigma(0)}, v_{\sigma(1)}, \ldots, v_{\sigma(n)}, h_{\sigma(0)}, h_{\sigma(1)}, \ldots, h_{\sigma(n)})$, $I_f(\pi_l(s_l)) = 0$, $m_r = \frac{(n-1)}{4} - \frac{k}{2n}$, $s_r = \infty$, $\pi_r(s_r) = (h_{\sigma(0)}, h_{\sigma(1)}, \ldots, h_{\sigma(n)}, v_{\sigma(0)}, v_{\sigma(1)}, \ldots, v_{\sigma(n)})$, $I_f(\pi_r(s_r)) = \frac{n(n-1)}{2}$ and $I_f(\pi(s^*)) = k$. It is easy to check that this initial condition satisfies $(I1)$ and $(I2)$. We also initialize the algorithm by setting the predecessor $pred(h_j)$ of each h_j in the set $Q_j = \{h_0, h_1, \ldots, h_j\}$ in $O(n \log n)$ time, where $pred(h_j) = \max\{h_i | h_i < h_j, i < j\}$ for each j. To find $pred(h_j)$ for each h_j, we can maintain a balanced binary tree $T(Q_j)$ dynamically such that we can do predecessor query on the node h_j in $O(\log n)$ time and then we can insert h_{j+1} into $T(Q_j)$ to obtain $T(Q_{j+1})$ in $O(\log n)$ time.

Adopting Cole's technique, we will decide for each new value s generated from the AKS sorting network if (s_l, s) or (s, s_r) is the *winning interval*, which contains s^*, and maintain the invariant conditions $(I1)$ and $(I2)$ for the winning interval. To do so, we need the following four subroutines, each taking $O(n)$ time. The *left reblocking* subroutine constructs an m_l-block left-compatible permutation $\pi_l(s)$ at s for which $I_f(\pi_l(s)) - I_f(\pi_l(s_l)) \leq 2m_l n$ holds. As we will show later, when $I_f(\pi_l(s)) - I_f(\pi_l(s_l)) > 2m_l n$, (s, s_r) cannot be the winning interval. Similarly the right reblocking subroutine constructs an m_r-block right-compatible permutation $\pi_r(s)$ at s for which $I_f(\pi_r(s_r)) - I_f(\pi_r(s)) \leq 2m_r n$ holds, and as we will also show later, when $I_f(\pi_r(s_r)) - I_f(\pi_r(s)) > 2m_r n$, (s_l, s) cannot be the winning interval. The *left halving*, and *right halving* subroutines are used to construct from an m_l-block left-compatible permutation $\pi_l(s)$ at s an $\frac{m_l}{2}$-block left-compatible permutation $\pi_l(s)$, and from an m_r-block right-compatible permutation $\pi_l(s)$ at s an $\frac{m_r}{2}$-block right-compatible $\pi_r(s)$, respectively.

For each new s, we first do left reblocking m_l and right reblocking m_r at s to construct $\pi_l(s)$ and $\pi_r(s)$ respectively. We distinguish three cases:

Case 1. $I_f(\pi_l(s)) - I_f(\pi_l(s_l)) > 2m_l n$: (s_l, s) will be the winning interval.
If the winning interval (s_l, s) does not satisfy $(I1)$ and $(I2)$, then we will do the right halving $\frac{m_r}{2^1}$, $\frac{m_r}{2^2}$, ... until $\frac{m_r}{2^t}$ such that $(I1)$ and $(I2)$ hold for (s_l, s).

Case 2. $I_f(\pi_r(s_r)) - I_f(\pi_r(s)) > 2m_r n$: (s, s_r) will be the winning interval.
If the winning interval (s, s_r) does not satisfy $(I1)$ and $(I2)$, then we will do the left halving $\frac{m_l}{2^1}, \frac{m_l}{2^2}, \ldots$ until $\frac{m_l}{2^t}$ such that $(I1)$ and $(I2)$ hold for (s, s_r).

Case 3. $I_f(\pi_l(s)) - I_f(\pi_l(s_l)) \leq 2m_l n$ and $I_f(\pi_r(s_r)) - I_f(\pi_r(s)) \leq 2m_r n$: We cannot decide the winning interval as yet.
We will do the left halving and the right halving alternately $\frac{m_l}{2^1}, \frac{m_r}{2^1}, \frac{m_l}{2^2}, \frac{m_r}{2^2}, \ldots$ until either $\frac{m_l}{2^t}$ or $\frac{m_r}{2^t}$, for some t, such that either (s, s_r) or (s_l, s) satisfies both $(I1)$ and $(I2)$. In the former case, (s, s_r) will be the winning interval, and in the latter, (s_l, s) will be the winning interval.

Having decided the winning interval, we can decide the relative ordering of s and s^*. So can we decide the relative ordering of relevant s_i's and s^*, where s was the weighted median of s_i's such that $sign(s - s_i) = sign(s^* - s)$. Then the above procedure repeats for each subsequent value s.

The algorithm continues to make approximations until $m_l < 10$ and $m_r < 10$. If $m_l < 10$ and $m_r < 10$, we have a winning interval (s_l, s_r) which contains s^* and $O(n)$ feasible intersection points. Let k' be the total number of feasible intersection points in $(-\infty, s_l]$ which can be obtained by the counting subroutine in Lemma 1. Then, we can enumerate all feasible intersection points in the winning interval (s_l, s_r) in $O(n \log n + n) = O(n \log n)$ time by the enumerating subroutine in Lemma 2, and select from those feasible intersection points the $(k - k')$th feasible intersection point with sum s^* by using any standard selection algorithm in $O(n)$ time.

Lemma 2 (*[15], Lemma 1*). *Given a sequence A of n real numbers $a_1, a_2, \ldots, a_n$ and two real numbers s_ℓ, s_r with $s_\ell \leq s_r$, it takes $O(n)$ space and $O(n \log n + h)$ time, where h is the output size, to find all segments $A(i, j)$, $1 \leq i \leq j \leq n$, among all $\frac{n(n-1)}{2}$ segments such that their sums $s(i, j)$ satisfy $s_\ell \leq s(i, j) \leq s_r$.*

We now give in more detail about the left reblocking, right reblocking, left halving and right halving subroutines and analyze their complexities.

We shall present the left reblocking subroutine as an example since the right reblocking subroutine can be done similarly. The left reblocking subroutine will either construct an m_l-block left-compatible permutation $\pi_l(s)$ for which $I_f(\pi_l(s)) - I_f(\pi_l(s_l)) \leq 2m_l n$ holds or output "fail" otherwise. Given an m_l-block left-compatible permutation $\pi_l(s_l)$, we obtain an m_l-block left-compatible permutation $\pi_l(s)$ at s for some $s > s_l$ only if $I_f(\pi_l(s)) - I_f(\pi_l(s_l)) \leq 2m_l n$.

Suppose we have an m_l-block S of H at s_l, an m_l-block T of V at s_l, an m_l-block left-compatible permutation $\pi_l(s_l)$, and $I_f(\pi_l(s_l))$. And we also maintain $\mu_l(v_j(s_l))$ to be $\min\{h_i | h_i \in S_t, i < j\}$ for each $v_j \in T_t$. We want to find an m_l-block left-compatible permutation $\pi_l(s)$, and $I_f(\pi_l(s))$ for some $s > s_l$. Let us process the lines one by one according to the order $v_0, h_0, v_1, h_1, \ldots, v_n, h_n$ to construct $\pi_l(s)$ at s. We will maintain a linked list of stacks, $\mathcal{S} = \{\mathcal{S}_0, \mathcal{S}_1, \ldots, \mathcal{S}_q\}$, each of which containing between $m_l/2$ and m_l elements of H such that the following property holds: if $h_a \in \mathcal{S}_i$, $h_b \in \mathcal{S}_j$ and $i < j$, then $h_a < h_b$. For each stack $\mathcal{S}_t$, we keep two counters: the total number $\nu(\mathcal{S}_t)$ of elements in $\mathcal{S}_t$ and the smallest element $l(\mathcal{S}_t)$ in $\mathcal{S}_t$ processed so far. Initially the first $m_l/2$ elements of H are in $\mathcal{S}_0$, $\nu(\mathcal{S}_0) = m_l/2$ and $l(\mathcal{S}_0) = \min\{h_0, h_1, \ldots, h_{m_l/2-1}\}$. The linked list of stacks $\mathcal{S}$ allows us to construct an m_l-block S of H and an m_l-block T of V at s when the subroutine terminates. We also maintain $\mu_l(v_j(s))$ for each j. Initially $\mu_l(v_j(s))$ is set to be $\mu_l(v_j(s_l))$. We also maintain a counter I_f such that it is $I_f(\pi_l(s))$ when the subroutine terminates. Initially I_f is set to be $I_f(\pi_l(s_l))$.

While processing v_i, we first let $\mathcal{S}_j$ be the stack such that $\mu_l(v_i(s)) \in \mathcal{S}_j$ and while $v_i(s) \geq l(\mathcal{S}_{j+1})$ we then do the following three steps: we set $\mu_l(v_i(s))$ to be $l(\mathcal{S}_{j+1})$, set I_f to be $I_f + \nu(\mathcal{S}_j)$ and set j to be $j + 1$, until $v_i(s) < l(\mathcal{S}_{j+1})$. Note that we assume here $\mathcal{S}_{j+1}$ is the successor of $\mathcal{S}_j$ in the linked list. (See the pseudocode lines 5–8.)

While processing h_i, we will insert h_i into some stack in the linked list $\mathcal{S}$ as follows. If $pred(h_i)$ exists, we will insert h_i into $\mathcal{S}_t$ which is the stack such that $pred(h_i) \in \mathcal{S}_t$. If $pred(h_i)$ does not exist, we will insert h_i into $\mathcal{S}_0$ and update $l(\mathcal{S}_0)$ to be h_i. (See the pseudocode lines 9-14.) After inserting h_i into some stack $\mathcal{S}_r$, if $\mathcal{S}_r$ has $m_l + 1$ elements we will split $\mathcal{S}_r$ into $\mathcal{S}_{r'}$ and $\mathcal{S}_{r''}$ containing $m_l/2$ and $m_l/2 + 1$ elements respectively, where $\mathcal{S}_{r'}$ appears before $\mathcal{S}_{r''}$ in this linked list. To do so, we compute the median h_m in $\mathcal{S}_r$, and lines $h_r \in \mathcal{S}_r$ such that $h_r < h_m$ are reinserted into the new stack $\mathcal{S}_{r'}$, and the remainder are reinserted into the new stack $\mathcal{S}_{r''}$. Note that $\nu(\mathcal{S}_{r'})$ is reset to $m_l/2$, $\nu(\mathcal{S}_{r''})$ is reset to $m_l/2 + 1$, and the smallest elements $l(\mathcal{S}_{r'})$ and $l(\mathcal{S}_{r''})$ are reset accordingly. (See the pseudocode lines 15–25.)

A detailed description of the left reblocking subroutine is shown in the pseudocode below. The whole procedure can be done in $O(n)$ time if $I_f(\pi_l(s)) - I_f(\pi_l(s_l)) \leq 2m_l n$. This can be easily seen by the fact that the total processing time is proportional to the number of times each v_i steps up the stacks, which is $O(n)$, since each time this happens, its rank is incremented by $O(m_l)$, and the difference of the ranks between $I_f(\pi_l(s))$ and $I_f(\pi_l(s_l))$ is at most $2m_l n$. And once we find that $I_f(\pi_l(s)) - I_f(\pi_l(s_l)) > 2m_l n$, we halt and output "fail". The time taken by a split operation of a stack is also $O(m_l)$ and the number of split operations is bounded by the total number of stacks, which is $O(\frac{n}{m_l})$.

Subroutine LeftReblocking(s_l, m_l, s)
Input: An m_l-block left-compatible permutation $\pi_l(s_l)$.
Output: An m_l-block left-compatible permutation $\pi_l(s)$.

1. **for** $i = 0$ **to** n **do** $\mu_l(v_i(s)) \leftarrow \mu_l(v_i(s_l))$;
2. $I_f \leftarrow I_f(\pi_l(s_l))$; $g \leftarrow 0$;
3. insert $h_0, h_1, \ldots, h_{m_l/2-1}$ into $\mathcal{S}_0$; $\nu(\mathcal{S}_0) \leftarrow m_l/2$;
   $l(\mathcal{S}_0) \leftarrow \min\{h_0, h_1, \ldots, h_{m_l/2-1}\}$; $q \leftarrow 0$; $l(\mathcal{S}_{q+1}) \leftarrow \infty$;

4. **for** $i = m_l/2$ **to** n **do**
   /* lines 5-8 process v_i, line 8 checks $I_f(\pi_l(s)) - I_f(\pi_l(s_l)) > 2m_l n$ */
5. let $\mathcal{S}_j$ be the stack such that $\mu_l(v_i(s)) \in \mathcal{S}_j$;
6. **while** $v_i(s) \geq l(\mathcal{S}_{j+1})$
7. $\mu_l(v_i(s)) \leftarrow l(\mathcal{S}_{j+1})$; $I_f \leftarrow I_f + \nu(\mathcal{S}_j)$; $g \leftarrow g + \nu(\mathcal{S}_j)$; $j \leftarrow j + 1$;
8. **if** $g > 2m_l n$ **then return** fail;
   /* lines 9-25 process h_i */
9. **if** $pred(h_i)$ exists
10. **then**
11. let $\mathcal{S}_t$ be the stack such that $pred(h_i) \in \mathcal{S}_t$;
12. insert h_i into $\mathcal{S}_t$; $\nu(\mathcal{S}_t) \leftarrow \nu(\mathcal{S}_t) + 1$; $r \leftarrow t$;
13. **else**
14. insert h_i into $\mathcal{S}_0$; $\nu(\mathcal{S}_0) \leftarrow \nu(\mathcal{S}_0) + 1$; $r \leftarrow 0$; $l(\mathcal{S}_0) \leftarrow h_i$;
15. **if** $\nu(\mathcal{S}_r) == m_l + 1$ **then**
16. $\nu(\mathcal{S}_{r''}) = \nu(\mathcal{S}_{r'}) \leftarrow 0$; $l(\mathcal{S}_{r''}) = l(\mathcal{S}_{r'}) \leftarrow \infty$; $q \leftarrow q + 1$; $l(\mathcal{S}_{q+1}) \leftarrow \infty$;
17. let h_m be the median in $\mathcal{S}_r$;
18. **for** each $h_t \in \mathcal{S}_r$ such that $h_t < h_m$ **do**
19. insert h_t into $\mathcal{S}_{r'}$; $\nu(\mathcal{S}_{r'}) \leftarrow \nu(\mathcal{S}_{r'}) + 1$;
20. **if** $h_t < l(\mathcal{S}_{r'})$ **then** $l(\mathcal{S}_{r'}) \leftarrow h_t$;
21. **for** each $h_t \in \mathcal{S}_r$ such that $h_t \geq h_m$ **do**
22. insert h_t into $\mathcal{S}_{r''}$; $\nu(\mathcal{S}_{r''}) \leftarrow \nu(\mathcal{S}_{r''}) + 1$;
23. **if** $h_t < l(\mathcal{S}_{r''})$ **then** $l(\mathcal{S}_{r''}) \leftarrow h_t$;
    /* lines 24-25 can be implemented by linked list in constant time */
24. renumber $\mathcal{S}_{r+1}, \mathcal{S}_{r+2}, \ldots, \mathcal{S}_{q-1}$ to be $\mathcal{S}_{r+2}, \mathcal{S}_{r+3}, \ldots, \mathcal{S}_q$;
25. renumber $\mathcal{S}_{r'}, \mathcal{S}_{r''}$ to be $\mathcal{S}_r, \mathcal{S}_{r+1}$;
    /* lines 26-31 construct m_l-block left-compatible permutation $\pi_l(s)$, $I_f(\pi_l(s))$, m_l-block S of H at s, m_l-block T of V at s */
26. **for** $i = 0$ **to** q **do**
27. $S_i \leftarrow null$; $T_i \leftarrow null$;
28. **for** each v_j such that $\mu_l(v_j(s)) \in \mathcal{S}_i$ **do** insert v_j into list T_i;
29. **for** each element h_j in $\mathcal{S}_i$ **do** insert h_j into list S_i;
30. Concatenate the list $T_0, S_0, T_1, S_1, \ldots, T_q, S_q$ to obtain $\pi_l(s)$;
31. $I_f(\pi_l(s)) \leftarrow I_f$; $S \leftarrow \{S_0, S_1, \ldots, S_q\}$; $T \leftarrow \{T_0, T_1, \ldots, T_q\}$;
32. **return** $\pi_l(s)$;

Thus we have the following.

Lemma 3 (*Reblocking*). *Given an m-block left-compatible permutation $\pi_l(s_l)$ with approximation rank $I_f(\pi_l(s_l))$, we can compute in $O(n)$ time an m-block left-compatible permutation $\pi_l(s)$ with approximation rank $I_f(\pi_l(s))$ for any $s > s_l$ for which $I_f(\pi_l(s)) - I_f(\pi_l(s_l)) \leq cmn$ for some constant c.*

Since the left halving and right halving subroutines are similar, we describe only the left halving subroutine as follows. Given an m_l-block left-compatible permutation $\pi_l(s)$, the left halving subroutine is to find a $\frac{m_l}{2}$-block left-compatible permutation $\pi_l(s)$. As before, suppose we have an m_l-block $S = \{S_0, S_1, \ldots, S_q\}$ of H at s, an m_l-block $T = \{T_0, T_1, \ldots, T_q\}$ of V at s, an m_l-block left-compatible permutation $\pi_l(s)$, $I_f(\pi_l(s))$, and maintain $\mu_l(v_j(s))$ for each j. We want to find a $\frac{m_l}{2}$-block left-compatible permutation $\pi_l(s)$ and $I_f(\pi_l(s))$. We will obtain $\frac{m_l}{2}$-block left-compatible permutation $\pi_l(s)$ by partitioning each S_i in S into two subsets S_i' and S_i'', S_i' contains the elements smaller than h_{m_i}, S_i'' contains the elements greater than or equal to h_{m_i}, where h_{m_i} is the median of S_i.

As before, we process the lines one by one according to the order $v_0, h_0, v_1, h_1, \ldots, v_n, h_n$ to construct a $\frac{m_l}{2}$-block left-compatible permutation $\pi_l(s)$ at s. We will maintain a linked list of stacks, $\mathcal{S} = \{\mathcal{S}_0, \mathcal{S}_1, \ldots, \mathcal{S}_{2q}\}$, each of them containing between $m_l/4$ and $m_l/2$ elements of H. We will insert the lines h_i's in S_j' into the stack $\mathcal{S}_{2j}$ and the lines h_i's in S_j'' into the stack $\mathcal{S}_{2j+1}$. For each stack $\mathcal{S}_j$, we keep two counters: the total number of elements $\nu(\mathcal{S}_j)$ in $\mathcal{S}_j$, the smallest element $l(\mathcal{S}_j)$ in $\mathcal{S}_j$ processed so far.

While processing v_i, we first let T_j be the set such that $v_i \in T_j$ and if $v_i(s) \geq h_{m_j}(s)$ and $v_i(s) \geq l(\mathcal{S}_{2j+1})$ then we set I_f to be $I_f + \nu(\mathcal{S}_{2j})$ and set $\mu_l(v_i(s))$ to be $l(\mathcal{S}_{2j+1})$. (See the pseudocode lines 7–9.)

While processing h_i, we first let S_j be the set such that $h_i \in S_j$ and if $h_i < h_{m_j}$ we insert h_i into the stack $\mathcal{S}_{2j}$ and then update $\nu(\mathcal{S}_{2j})$ and $l(\mathcal{S}_{2j})$ accordingly, otherwise we insert h_i into the stack $\mathcal{S}_{2j+1}$ and then update $\nu(\mathcal{S}_{2j+1})$ and $l(\mathcal{S}_{2j+1})$ accordingly. (See the pseudocode lines 10–17.)

A detailed description of the left halving subroutine is shown in the pseudocode below. The entire procedure can be done in $O(n)$ time since each v_i steps up at most one block.

Subroutine LeftHalving(s, m_l)
Input: An m_l-block left-compatible permutation $\pi_l(s)$.
Output: An $\frac{m_l}{2}$-block left-compatible permutation $\pi_l(s)$.

1. $I_f \leftarrow I_f(\pi_l(s))$;
2. **for** $i = 0$ **to** q **do**
3. find h_{m_i} to be the median in set S_i;
4. **for** $i = 0$ **to** $2q$ **do**
5. $\nu(\mathcal{S}_i) \leftarrow 0$; $l(\mathcal{S}_i) \leftarrow null$;
6. **for** $i = 0$ **to** n **do**
   /* lines 7-9 process v_i */
7. let T_j be the set such that $v_i \in T_j$;
8. **if** $v_i(s) \geq h_{m_j}(s)$ and $l(\mathcal{S}_{2j+1}) \neq null$ and $v_i(s) \geq l(\mathcal{S}_{2j+1})$
9. **then** $\mu_l(v_i(s)) \leftarrow l(\mathcal{S}_{2j+1})$; $I_f \leftarrow I_f + \nu(\mathcal{S}_{2j})$;
   /* lines 10-17 process h_i */
10. let S_j be the set such that $h_i \in S_j$;
11. **if** $h_i < h_{m_j}$
12. **then**
13. insert h_i into $\mathcal{S}_{2j}$; $\nu(\mathcal{S}_{2j}) \leftarrow \nu(\mathcal{S}_{2j}) + 1$;
14. **if** $l(\mathcal{S}_{2j}) = null$ or $h_i < l(\mathcal{S}_{2j})$ **then** $l(\mathcal{S}_{2j}) \leftarrow h_i$;
15. **else**
16. insert h_i into $\mathcal{S}_{2j+1}$; $\nu(\mathcal{S}_{2j+1}) \leftarrow \nu(\mathcal{S}_{2j+1}) + 1$;
17. **if** $l(\mathcal{S}_{2j+1}) = null$ or $h_i < l(\mathcal{S}_{2j+1})$ **then** $l(\mathcal{S}_{2j+1}) \leftarrow h_i$;
   /* lines 18-24 construct $\frac{m_l}{2}$-block left-compatible permutation $\pi_l(s)$, $I_f(\pi_l(s))$, $\frac{m_l}{2}$-block S of H at s, $\frac{m_l}{2}$-block T of V at s */
18. **for** $i = 0$ **to** $2q$ **do**
19. $S_i \leftarrow null$; $T_i \leftarrow null$;
20. **for** each v_j such that $\mu_l(v_j(s)) \in \mathcal{S}_i$ **do** insert v_j into list T_i;
21. **for** each element h_j in $\mathcal{S}_i$ **do** insert h_j into list S_i;
22. Concatenate the list $T_0, S_0, T_1, S_1, \ldots, T_{2q}, S_{2q}$ to obtain $\pi_l(s)$;
23. $I_f(\pi_l(s)) \leftarrow I_f$; $S \leftarrow \{S_0, S_1, \ldots, S_{2q}\}$; $T \leftarrow \{T_0, T_1, \ldots, T_{2q}\}$;
24. **return** $\pi_l(s)$;

Thus we have Lemma 4.

Lemma 4 (*Halving*). *Given an m-block left-compatible permutation $\pi_l(s)$ with approximation rank $I_f(\pi_l(s))$ for some s, we can compute in $O(n)$ time an $\frac{m}{2}$-block left-compatible permutation $\pi_l(s)$ with approximation rank $I_f(\pi_l(s))$.*

We now explain the algorithm, analyze its complexity. For each new value s from sorting network, we first do left reblocking m_l and right reblocking m_r at s. If left blocking fails, we have $I_f(\pi(s)) > I_f(\pi_l(s)) > I_f(\pi_l(s_l)) + 2m_l n > I_f(s^*)$. It implies $s > s^*$. That is, we can decide (s_l, s) is the winning interval. If m_r is not small enough such that (s_l, s) satisfies $(I1)$ and $(I2)$, we do right halving at s until both $(I1)$ and $(I2)$ hold. Similarly, if right blocking fails, we have $I_f(\pi(s)) < I_f(\pi_r(s)) < I_f(\pi_r(s_r)) - 2m_r n < I_f(s^*)$. It implies $s < s^*$. That is, we can decide (s, s_r) is the winning interval. If m_l is not small enough such that (s, s_r) satisfies $(I1)$ and $(I2)$, we do left halving at s until both $(I1)$ and $(I2)$ hold. If both left blocking and right blocking do not fail then we have $I_f(\pi_l(s_l)) + 2m_l n \geq I_f(\pi_l(s))$ and $I_f(\pi_r(s)) \geq I_f(\pi_r(s_r)) - 2m_r n$. It means that both m_l and m_r are not fine enough to decide the winning interval. We will do left halving and right halving at s alternately $\frac{m_l}{2^1}$, $\frac{m_r}{2^1}$, $\frac{m_l}{2^2}$, $\frac{m_r}{2^2}$, ... until $\frac{m_l}{2^t}$ or $\frac{m_r}{2^t}$ such that both conditions $(I1)$ and $(I2)$ hold either for (s, s_r) or for (s_l, s). In the former (s, s_r) will be the winning interval, and in the latter, (s_l, s) is the winning interval. After the winning interval is decided, we can answer the comparison question at s and the relevant s_i's of which s was the weighted median such that $sign(s - s_i) = sign(s^* - s)$. Then another new value s, if any, will be generated, and the above procedure repeats.

Since the algorithm will invoke $O(1)$ left blocking and right blocking subroutines at each parallel step to resolve all comparisons at this step, each taking $O(n)$ time, it totally takes $O(n)$ time at each step. The sorting network has depth $O(\log n)$, each parallel step requires $O(n)$, so the algorithm takes $O(n \log n)$ time to do left blocking and right blocking. But during the execution of the left or right blocking algorithm, since the approximation sometimes is not good enough to distinguish the relative ordering of s and s^*, we need to refine the approximation until we can decide relative ordering of s and s^*. The algorithm will at most invoke $O(\log n)$ left halving and right halving subroutines, each taking $O(n)$. It turns out that an amortized $O(n \log n)$ extra time is sufficient to refine approximations throughout the entire course of the algorithm. The correctness of this algorithm follows from the above discussion. Thus, we conclude with the following theorem.

Theorem 1. *The* Sum Selection Problem *can be solved in $O(n)$ space and $O(n \log n)$ time.*

The complete pseudocode of the algorithm follows.

Algorithm Sum Selection Problem.
Input: A set of lines $H = \{h_0, h_1, \ldots, h_n\}$ and $V = \{v_0, v_1, \ldots, v_n\}$ in $\mathbf{R}^2$ where $h_i : y = -s_i$ and $v_j : y = x - s_j$, and a positive integer k.
Output: The feasible intersection point $p_{i^*j^*} = (x_{i^*j^*}, y_{i^*j^*})$ such that $r(x_{i^*j^*}, X_f) = k$.

1. **if** $k < n$ **then return**;
2. Sort H and find the permutation σ satisfying $h_{\sigma(0)} \leq h_{\sigma(1)} \leq \cdots \leq h_{\sigma(n)}$;
3. $m_l \leftarrow \frac{k}{n}$; $s_l \leftarrow -\infty$; $\pi_l(s_l) \leftarrow (v_{\sigma(0)}, v_{\sigma(1)}, \ldots, v_{\sigma(n)}, h_{\sigma(0)}, h_{\sigma(1)}, \ldots, h_{\sigma(n)})$;
   $I_f(\pi_l(s_l)) \leftarrow 0$;
4. $m_r \leftarrow \frac{(n-1)}{4} - \frac{k}{2n}$; $s_r \leftarrow \infty$; $\pi_r(s_r) \leftarrow (h_{\sigma(0)}, h_{\sigma(1)}, \ldots, h_{\sigma(n)}, v_{\sigma(0)}, v_{\sigma(1)}, \ldots, v_{\sigma(n)})$;
   $I_f(\pi_r(s_r)) \leftarrow \frac{n(n-1)}{2}$;
   /* *pred* can be thought of as a global array accessible to all subroutines */
5. **for** $j = 0$ **to** n **do** $pred(h_j) \leftarrow \max\{h_i | h_i < h_j, i < j\}$;
6. **for** $j = 0$ **to** n **do** $\mu_l(v_j(s_l)) \leftarrow \min\{h_i | i < j\}$; $\mu_r(v_j(s_r)) \leftarrow \max\{h_i | i < j\}$;
7. **while** $m_l > 10$ **or** $m_r > 10$
8. get next s from AKS network
9. **if** s is not in (s_l, s_r)
10. **then** resolve s and the relevant s_i's such that $sign(s - s_i) = sign(s^* - s)$;
11. **else**
12. $m'_l \leftarrow m_l$; $\pi_l(s) \leftarrow$ LeftBlocking(s, s_l, m'_l);
13. $m'_r \leftarrow m_r$; $\pi_r(s) \leftarrow$ RightBlocking(s, s_r, m'_r);
14. **case 1:** LeftBlocking subroutine outputs "fail"
15. **while** (s_l, s) does not satisfy $(I1)$, $(I2)$
16. $\pi_r(s) \leftarrow$ RightHalving(s, m'_r); $m'_r \leftarrow \frac{m'_r}{2}$;
17. **case 2:** RightBlocking subroutine outputs "fail"
18. **while** (s, s_r) does not satisfy $(I1)$, $(I2)$
19. $\pi_l(s) \leftarrow$ LeftHalving(s, m'_l); $m'_l \leftarrow \frac{m'_l}{2}$;
20. **case 3:** LeftBlocking **and** RightBlocking do not output "fail"
21. **while** both (s_l, s) and (s, s_r) do not satisfy $(I1)$, $(I2)$
22. $\pi_l(s) \leftarrow$ LeftHalving(s, m'_l); $m'_l \leftarrow \frac{m'_l}{2}$;
23. $\pi_r(s) \leftarrow$ RightHalving(s, m'_r); $m'_r \leftarrow \frac{m'_r}{2}$;
24. **if** (s, s_r) satisfies $(I1)$ and $(I2)$ **then**
25. $s_l \leftarrow s$; $m_l \leftarrow m'_l$;
26. resolve s and the relevant s_i's such that $sign(s - s_i) = sign(s^* - s)$;
27. **if** (s_l, s) satisfies $(I1)$ and $(I2)$ **then**
28. $s_r \leftarrow s$; $m_r \leftarrow m'_r$;
29. resolve s and the relevant s_i's such that $sign(s - s_i) = sign(s^* - s)$;
30. $k' \leftarrow$ total number of feasible points in $(-\infty, s_l]$ by Lemma 1
31. $S \leftarrow$ the set of all feasible points in (s_l, s_r) by Lemma 2
32. **return** $s^* \leftarrow (k - k')$th element in S by any optimal selection algorithm

3. Algorithm for *k* Maximum Sums Problem

After obtaining the algorithm for the Sum Selection Problem, we can use it to obtain the algorithm for k Maximum Sums Problem directly. We have the following result.

Theorem 2. *The* k Maximum Sums Problem *can be solved in* $O(n)$ *space and* $O(n \log n + k)$ *time.*

Proof. Let $\ell = \frac{n(n-1)}{2} - k + 1$. We can run the algorithm of the Sum Selection Problem to obtain the ℓth smallest segment sum s_ℓ in $O(n \log n)$ time and then we can enumerate the k largest sum segments by the enumerating subroutine Lemma 2 in the interval $[s_\ell, \infty)$ in $O(n \log n + k)$ time. □

4. Conclusion

In this paper we have presented a deterministic algorithm for the Sum Selection Problem that runs in $O(n \log n)$ time. We then use it to give a more efficient algorithm for the k Maximum Sums Problem that runs in $O(n \log n + k)$ time. It is better than the previously best known result for the problem, but whether or not one can prove an $\Omega(n \log n)$ lower bound for the Sum Selection Problem is of great interest.

Acknowledgements

The authors would like to thank the reviewers for improving the presentation of this paper.

References

[1] R. Agrawal, T. Imielinski, A. Swami, Data mining using two-dimensional optimized association rules: Scheme, algorithms, and visualization, in: Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data, 1993, pp. 207–216.
[2] M. Ajtai, J. Komlós, E. Szemerédi, An $O(n \log n)$ sorting networks, Combinatorica 3 (1983) 1–19.
[3] S. Alk, G. Guenther, Application of broadcasting with selective reduction to the maximal sum subsegment problem, International Journal of High Speed Computating 3 (1991) 107–119.
[4] S.E. Bae, T. Takaoka, Algorithms for the problem of k maximum sums and a VLSI algorithm for the k maximum subarrays problem, in: 2004 International Symposium on Parallel Architectures, Algorithms and Networks, 2004, pp. 247–253.
[5] S.E. Bae, T. Takaoka, Improved algorithms for the *k*-maximum subarray problem, The Computer Journal 49 (3) (2006) 358–374.
[6] F. Bengtsson, J. Chen, Efficient algorithms for *k* maximum sums, Algorithmica 46 (2006) 27–41.
[7] J. Bentley, Programming pearls: Algorithm design techniques, Communications of the ACM 27 (9) (1984) 865–873.
[8] J. Bentley, Programming pearls: Algorithm design techniques, Communications of the ACM 27 (11) (1984) 1087–1092.
[9] H. Brönnimannm, B. Chazelle, Optimal slope selection via cuttings, Computational Geometry 10 (1998) 23–29.
[10] C.-H. Cheng, K.-Y. Chen, W.-C. Tien, K.-M. Chao, Improved algorithms for the k maximum-sums problems, Theoretical Computer Science 362 (2006) 162–170.
[11] R. Cole, J.S. Salowe, W.L. Steiger, E. Szemeredi, An optimal-time algorithm for slope selection, SIAM Journal on Computing 18 (4) (1989) 792–810.
[12] R. Cole, Slowing down sorign networks to obtain faster sorting algorithm, Journal of the Association for Computing Machinery 34 (1) (1987) 200–208.
[13] T. Fukuda, Y. Morimoto, S. Morishita, T. Tokuyama, Mining association rules between sets of items in large databases, in: Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data, 1996, pp. 13–23.
[14] D. Gries, A note on the standard strategy for developing loop invariants and loops, Science of Computer Programming 2 (1982) 207–214.
[15] T.-C. Lin, D.T. Lee, Randomized algorithm for the sum selection problem, Theoretical Computer Science 377 (1–3) (2007) 151–156.
[16] N. Megiddo, Applying parallel computation algorithms in the design of serial algorithm, Journal of the Association for Computing Machinery 30 (4) (1983) 852–865.
[17] K. Perumalla, N. Deo, Parallel algorithms for maximum subsequence and maximum subarray, Parallel Processing Letters 5 (1995) 367–373.
[18] K. Qiu, S. Alk, Parallel maximum sum algorithms on interconnection networks, Technical report no. 99-431, Jodrey school of computer science, Acadia University, Canada, 1999.
[19] D. Smith, Applications of a strategy for designing divide-and-conquer algorithms, Science of Computer Programming 8 (1987) 213–229.
[20] T. Takaoka, Efficient algorithms for the maximum dubarray problem by fistance matrix multiplication, in: Proceedings of the 2002 Australian Theory Symposium, 2002, pp. 189–198.
[21] H. Tamaki, T. Tokuyama, Algorithms for the maximum subarray problem based on matrix multiplication, in: Proceedings of the ninth annual ACM-SIAM symposium on discrete algorithms, 1998, pp. 446–452.